Algorithms
Computer Science 140 & Mathematics 168
Spring 2018
Homework 2
Due: Tuesday, January 30 at 11:59 PM

Under the Harvey Mudd Honor Code, this document is not to be shared.

Problem 1: Generalizing Mergesort [20 Points]

Researchers at the Pasadena Institute of Technology are interested in solving the problem of finding the maximum element in an array of length n. One simple way to do this is to just iterate through the array, keeping track of the largest element seen so far. When the iteration completes, we've found the largest element in the array. This algorithm takes time $O(n)$. The researchers at P.I.T. have proposed two recursive divide-and-conquer algorithms for finding the maximum element described below. For each algorithm, write the recurrence relation that describes its running time and then use the "work tree" method from class to derive its asymptotic running time as a function of n. (The method is called the "work tree" method, but it's just the table that keeps track of four columns: The number of nodes at this level of recursion, the problem size, the work per node, and the total amount of work at this level.) Please show your work.

1. The first algorithm divides the array into two equal halves and recursively finds the maximum in each half. Let the maximum in the left half be denoted by x and the maximum in the right half be denoted by y. The algorithm now computes and returns $\max(x, y)$.

2. The second algorithm divides the array into three equal thirds and recursively finds the maximum in each third. Let the three maximum values for the first, second, and last thirds be denoted x, y, and z, respectively. The algorithm now computes and returns $\max(x, y, z)$.

3. How do the asymptotic (that is, big-O) running times of these two algorithms compare to the $O(n)$ asymptotic running time of the simple "iterate through the array and keep track of the largest value" algorithm?

Solution: For the first algorithm, the recurrence relation is $T(n) = 2T(n/2) + c$ with base case $T(1) = c$. Notice that it takes constant time to compute the maximum of the result returned by the two recursive calls. You should show the work tree table. It solves to $O(n)$. Here's the analysis where we assume that $n = 2^k$.

Nodes	Problem Size	Work per Node	Total Work
1	2^k	c	c
2	2^{k-1}	c	$2c$
2^2	2^{k-2}	c	2^2c
$\ldots$	$\ldots$	$\ldots$	$\ldots$
2^k	2^{k-k}	c	2^kc

Adding up all this work gives $c(1 + 2 + \ldots 2^k) = c(2^{k+1} - 1) \leq 2cn \in O(n)$.

For the second algorithm, the recurrence relation is $T(n) = 3T(n/3) + c$ with base case $T(1) = c$. Use $n = 3^k$ when solving this recurrence relation. It also solves to $O(n)$. Thus, both algorithms are asymptotically equivalent to the iterative algorithm.

Problem 2: Stoogesort! [20 Points]

Professors Curly, Mo, and Larry of the Pasadena Institute of Technology have proposed the following sorting algorithm: First sort the first two-thirds of the elements in the array. Next sort the last two thirds of the array. Finally, sort the first two thirds again. (Notice that this algorithm is similar to Mergesort except that it uses three recursive calls rather than two and there is no merging step! As a consequence, this algorithm is very easy to implement!) The code is given below, but it's only there for reference; it suffices to just think about the description of the algorithm above.

```
Stoogesort (A,i,j)
begin
    if A[i] > A[j] then
        swap A[i] and A[j]
    if i + 1 ≥ j then
        return
    k = ⌊(j − i + 1)/3⌋.
    Stoogesort(A, i, j − k)     Comment: Sort first two-thirds.
    Stoogesort(A, i + k, j)     Comment: Sort last two-thirds.
    Stoogesort(A, i, j − k)     Comment: Sort first two-thirds again!
end
```

a. Give an informal but convincing explanation (not a rigorous proof by induction) of why the approach of sorting the first two-thirds of the array, then sorting the last two-thirds of the array, and then sorting again the first two-thirds of the array yields a sorted array. A few well-chosen sentences should suffice here.

b. Find a recurrence relation for the worst-case running time of Stoogesort. To simplify your recurrence relation, you may assume each of the recursive calls is on a portion of the array that is *exactly* two-thirds the length of the original array.

c. Next, solve the recurrence relation using the work tree method. **Show all of your work.** In your analysis, it will be convenient to choose n to be c^k for some fixed constant c. (For

example, we used $c = 2$ when analyzing Mergesort. Here you will want to use a different value of c. The value of c that you choose might not even be an integer! As we'll see in class, this is valid as long as $c > 1$.)

d. How does the worst-case asymptotic running time of Stoogesort compare with the worst-case asymptotic running time of Mergesort?

Solution:

a. A typical solution will go something like this. "After sorting the first two-thirds of the array, the elements that reside in the lowest third of the array *at this moment* cannot possibly belong in the upper third of the *correctly sorted array* because we have already found $n/3$ elements that are larger. Thus, after sorting the last two-thirds of the array, the last one-third of the array at this moment necessarily contains the largest third of the elements overall and they are in sorted order! The lowest two-thirds of the array is not necessarily in sorted order yet, but sorting the first two-thirds again ensures that the lowest two-thirds is now sorted.

b. $T(n) = 3T(\frac{2}{3}n) + c$ and $T(1) = c$. (You can also define $T(1) = 1$ or $T(1)$ equals some other constant.)

c. Let $n = (3/2)^k$ for convenience. Now, we use the work tree method:

Nodes	Problem Size	Work per Node	Total Work
1	$(3/2)^k$	c	c
3	$(3/2)^{k-1}$	c	$3c$
3^2	$(3/2)^{k-2}$	c	3^2c
$\cdots$	$\cdots$	$\cdots$	$\cdots$
3^k	$(3/2)^{k-k}$	c	3^kc

Thus,

$$T(n) = c(1 + 3 + \cdots + 3^k) = c\frac{3^{k+1} - 1}{3 - 1} \in \Theta(3^k)$$

However,

$$\Theta(3^k) = \Theta(3^{\log_{3/2} n}) = n^{\log_{3/2} 3} = \Theta(n^{2.709\ldots})$$

d. Stoogesort is slower.

Problem 3: Robot Testing! [30 Points]

The famous roboticist, Professor Dodd S. Zachovichsky, has n supposedly identical robots that – in principle – are capable of testing each other. The professor's test jig accommodates two robots at a time. When the jig is loaded, each robot tests the other and reports whether it is good or bad. A good robot always reports accurately whether the other robot is good

or bad, but the answer of a bad robot cannot be trusted. Thus, the four possible outcomes of a test are as follows:

Robot A says	Robot B says	Conclusion
B is good	A is good	Both good, or both bad
B is good	A is bad	At least one is bad
B is bad	A is good	At least one is bad
B is bad	A is bad	At least one is bad

a. Consider the problem of finding a single good robot from among n robots, assuming that more than $\frac{n}{2}$ of the robots are good. Show that $\lfloor \frac{n}{2} \rfloor$ pairwise tests are sufficient to reduce the problem to one that is about half the size (or smaller!). (You might find it convenient to begin by examining the case that n is even. Once, you've dealt with that case, the case that n is odd requires just a bit more work.)

b. Using the result of part (a) above, show that the good robots can be identified with $O(n)$ pairwise tests, assuming that more than $\frac{n}{2}$ of the robots are good. Explain clearly why your algorithm identifies all of the good robots. Then give and solve the recurrence that describes the number of tests.

Solution:

a. The idea here is to pair up the elements and let them test one another in pairs. We'll begin with the assumption that n is even. A given pair will produce one of the following outputs from a pairwise test: "good-good", "good-bad" (or vice-versa), or "bad-bad". For every "good-good" pair, we keep one of those two and throw out the other (arbitrarily, since we can't distinguish these). All other robots are thrown out. This results in a subset of the original set that is at most half the size of the original because we keep at most one robot from each pair. Now, we must show that in the new subset, more than half of the robots are still good. Here's why: In the original set, there were more good robots than bad. If a robot is actually good, let's call it a G-robot and if it's actually bad let's call it a B-robot. There must be some G-G pairs by the pigeonhole principle (there are more G's that B's, so some G's must pair up). Moreover, we will throw both robots away from a G-B pair because at least one of those robots will say "bad." So, the only robots that we will keep are one from every G-G pair and possibly one from each B-B pair (if they both lie and say "good"). But, since there are more G-G pairs than B-B pairs, there will be more good robots kept than bad ones. Thus, in $n/2$ comparisons we have reduced our original set into one that is at most half the size and maintains the property that there are still more good than bad robots.

   Next, we turn to the case that n is odd. In this case, one robot is left unpaired in our pairwise testing procedure. If the number of "good-good" outputs is odd, we simply

discard the unpaired robot. If the number of "good-good" outputs is even, we keep the unpaired robot in the next iteration of the algorithm. We show that this maintains the invariant that the next set of robots contains more good robots than bad ones. First, consider the case that we observe an odd number of "good-good" outputs and we denote this odd number by $2k + 1$. Then, there must be either more G-G pairs or more B-B pairs. If there are more B-B pairs then there are at least $k+1$ B-B pairs and thus at least $2(k + 1) = 2k + 2$ bad robots in that group and, thus, at most $2k$ good robots in that group. Thus, the number of bad robots exceeds the number of good robots by at least 2. Even if the unpaired robot is good, this still results in more bad robots than good robots in total, which is a contradiction. Thus, there must be more G-G pairs than B-B pairs in this group. Therefore, discarding the unpaired robot maintains the invariant. Finally, consider the remaining case that the number of "good-good" outputs is even and denote this even number by $2k$. At least k pairs must be G-G, since otherwise there are more bad robots than good robots in total. If there are exactly k G-G pairs, then the unpaired robot must be good in order for there to be more good robots than bad. And, if there are more than k G-G pairs then the number of good robots in that set exceeds the number of bad robots by at least $2(k + 1) - 2(k - 1) = 4$ and thus the invariant is maintained even if the unpaired robot is bad.

b. We repeat the process above until we are left with one good robot. Then, we use that robot to test all of the other robots. The recurrence relation for the worst case number of comparisons that we need to make is $T(n) = T(n/2) + n/2$ which solves to $\Theta(n)$.

Problem 4: Pasadenium! [30 Points]

Researchers at the Pasadena Institute of Technology are experimenting with a super-strong new material called Pasadenium. To test its strength, blocks of Pasadenium will be dropped from various floors of a tall building with n floors. A Pasadenium block is said to have *least breaking floor* (LBF) k if it will break when dropped from floor k but will not break if dropped from any lower floor. The LBF is defined to be $n + 1$ if it will not break from floor n (the uppermost floor).

Your task is to determine the LBF of a block of Pasadenium. Since it's a very expensive material, you are given just two blocks. Fortunately, any two blocks of Pasadenium have the same LBF.

It is easy to see that if you only had one block, you would have to drop the block $\Theta(n)$ times in the worst case to determine the LBF (successively drop the block from floor 1, floor 2, ..., floor n).

In this problem we will determine how to best utilize two blocks to determine the LBF using the least number of drops. Rather than expressing the solution as "Given a building with n floors, I can compute the LBF in at most $f(n)$ drops" we will instead express the solution *inversely* as "With d drops, I can compute the LBF of a building with as many as $g(d)$ floors." Here's your task:

a. Given two blocks of Pasadenium and no more than d drops, for how tall a building can you guarantee finding the LBF? Describe your algorithm in clear and concise English and give a closed form function $g(d)$ for the maximum height of the building (as a function of d) that your algorithm can handle.

b. Use induction on d to prove that your algorithm uses at most d block drops for a building with $g(d)$ floors.

c. Prof. I. Lai is skeptical. Write a short memo why he should believe that your algorithm is optimal; that is, why *any algorithm* that uses 2 blocks and at most d drops cannot guarantee determining the LBF for a building taller than the $g(d)$ that you gave in part (a) of this problem. A few sentences of intuitive explanation suffice here. (If you wish, you're welcome to prove this rigorously – it's not too hard to turn the informal explanation into a formal proof.)

Solution:

a. Let $g(d) = 1 + 2 + \cdots + d = d(d+1)/2$.

The algorithm works as follows: Our first drop is from floor d. If the block breaks from floor d, we know that the LBF is at floor d or below. So, with our one remaining block, we start at the first floor and move upwards until we find the LBF. However, if the block doesn't break from floor d, we know that the LBF is at floor $d+1$ or above. So, we can consider the upper part of the building from floors $d+1$ to $g(d)$ to be a new smaller building and start over on that smaller building with our two blocks.

b. We use induction on d to prove that this algorithm uses at most d block drops for a building with $g(d)$ floors. When $d = 1$, $g(d) = 1$ and clearly we can find the LBF of a one-story building with one block drop using our algorithm. The algorithm simply drops its one block on the first and only floor and determines whether or not the block breaks. Our induction hypothesis is that the algorithm works correctly for d (that is, it uses d block drops for a building of height $g(d)$) and in the induction step we wish to show that the algorithm works correctly for $d+1$ (that is, that it uses $d+1$ drops for a building of height $g(d+1)$). So, consider a building of height $g(d+1)$. Our algorithm drops the first block at floor $d+1$. If it breaks, there are d floors below and we test them from the bottom up, using at most d more block drops, for a total of $d+1$ drops. If, however, the block does not break at floor $d+1$, the remaining part of the building has floors $g(d+1) - (d+1) = 1 + \cdots + d = g(d)$ floors. So, we're left with an upper part of the building of $g(d)$ floors, we still have our two blocks, and the induction hypothesis asserts that we can find the LBF of $g(d)$ floors with d block drops. So, again, we use a total of at most $d+1$ drops. Q.E.D.

c. We can sketch the correctness by arguing that any optimal algorithm must agree with us on the first drop. If an optimal algorithm attempts to make the first drop from a floor

higher than d, then if that first brick breaks the algorithm will be unable to find the LBF. On the other hand, if the first drop is lower than floor d, then it can't do as well as our algorithm because it dropped the first block too low. Thus, an optimal algorithm must agree with our algorithm on the first drop and, repeating this logic, must agree with us on every successive drop.

We can rigorously prove this claim as follows: We will prove that any algorithm for computing the LBF using two blocks of Pasadenium on a building with $g(d) + 1$ floors requires more than d block drops. We first show that with *one block* of Pasadenium and d drops, we can handle a building with at most d floors. Let $f(d)$ be the maximum number of floors for which we can compute the LBF with one block and at most d drops. The crucial observation is that the first drop must be from floor 1. If the first drop is from a floor $j > 1$, then if the block breaks, we are unable to compute the LBF since it could be any floor less than j. If the block does not break, we must solve the same problem with $d - 1$ drops, so $f(d) \leq 1 + f(d - 1)$. Since $f(1) = 1$, this implies that $f(d) \leq d$ for all d.

Similarly, let $h(d)$ denote the maximum number of floors for which the LBF can be computed with two blocks and at most d drops. We observe that, for any algorithm, the first drop must be from a floor no higher than d: If we drop the first block from floor $j > 1$ and it breaks, then we are left with the problem of determining the LBF of a building of height $j - 1$ with only one block and at most $d - 1$ drops and this is only possible if $j - 1 \leq d - 1$ which implies that $j \leq d$. So, the first drop is necessarily from some floor $j \leq d$. If the block does not break, we are left with the problem of computing the LBF of a building with height $h(d) - j \geq h(d) - d$ using 2 blocks and at most $d - 1$ drops. Thus $h(d - 1) \geq h(d) - j \geq h(d) - d$ which implies that $h(d) \leq h(d - 1) + d$. Since $h(1) = 1$, this recurrence solves to $h(d) \leq d(d + 1)/2 = g(d)$. Q.E.D.